Monitoring Message-Passing Parallel Applications in the Grid with GRM and Mercury Monitor*

Norbert Podhorszki, Zoltán Balaton and Gábor Gombás

MTA SZTAKI, Budapest, H-1528 P.O.Box 63, Hungary
{pnorbert, balaton, gombasg}@sztaki.hu

Abstract. Application monitoring in the grid for parallel applications is hardly supported in recent grid infrastructures. There is a need to visualize the behavior of the program during its execution and to analyze its performance. In this paper the GRM application monitoring tool, the Mercury resource and job monitoring infrastructure and the combination of the two as a grid monitoring tool-set for message-passing parallel applications is described. By using them, one can see and analyze on-line the behavior and performance of the application.

1 Application monitoring in the grid

There are several parallel applications that are used on a single cluster or a supercomputer. As users get access to an actual grid they would like to execute their parallel applications on the grid instead of the local computing resource for several reasons. Local resources are always limited in size and availability. They might not be capable of executing an application with a given size of input data so the user should look for a larger resource. Another limitation is that the local resource is likely to be occupied by other users for days or weeks and we may need to run our application today.

In current grid implementations, we are already allowed to submit our parallel application to the grid and let it execute on a remote grid resource. However, those current grid systems are not able to give detailed information about our application during its execution except its status, like standing in a queue, running, etc. If we are interested in getting information about our application (how it is working, what its performance is) we realize that there is no way to collect such information. Performance analysis tools are not available yet for current grid systems.

Our target of research has been to provide a monitoring tool that is able to collect trace information about an – instrumented – parallel application executed on a remote grid resource. This goal is different from providing a monitoring tool for meta-computing applications running on several resources simultaneously, i.e. being distributed applications. The latter goal requires the tool to be able to monitor processes on more than one resource at the same time that brings other requirements for the tool.

Combining our GRM and Mercury Monitor tools we have achieved our goal and created an infrastructure that enables the user to collect performance information about

* The work described in this paper has been supported by the following grants: EU-DataGrid IST-2000-25182 and EU-GridLab IST-2001-32133 projects, the Hungarian Supergrid OMFB-00728/2002 project, the IHM 4671/1/2003 project and the grant OTKA T042459.

a parallel application and examine it the same way as it has been done on the local cluster in the past years.

2 Application monitoring with GRM

For monitoring parallel applications on a local resource, the GRM tool [4] can be used. GRM provides an instrumentation library for message passing parallel applications (such as MPI [2], PVM [3], P-GRADE [1]). GRM has been used to collect trace data from an application running on a cluster or a supercomputer.

GRM consists of Local Monitors (see Fig. 1) on each host where application processes are running and a Main Monitor at the host where the user is analyzing trace information. The Local Monitor creates a shared-memory buffer where application processes (the instrumentation functions in them) put trace records directly without any further notifications to the Local Monitor. This way, the intrusion of monitoring is as low as possible from a source code based instrumentation. The Main Monitor of GRM controls the work of the Local Monitors and collects the content of the trace buffers.

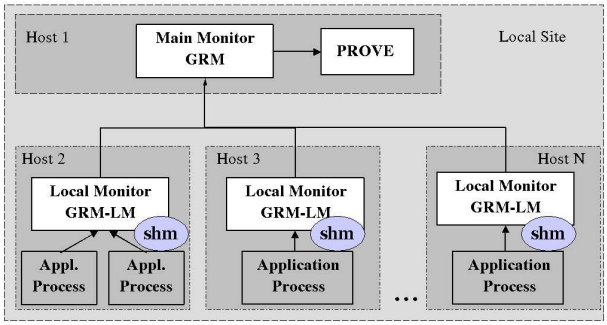

Fig. 1. Structure of GRM

For trace generation the user should first instrument the application with trace generation functions. In the general case the user would use the GRM instrumentation API and library to instrument her PVM or MPI application manually. However, if GRM is used as an integrated part of P-GRADE (i.e. the parallel application is generated from P-GRADE) the user does not need to know anything about the instrumentation library of GRM and its API. In this case P-GRADE automatically inserts the necessary instrumentation code according to the user's needs that can be specified completely graphically. (See the P-GRADE User's Manual [1].)

The strength of the GRM instrumentation is that besides the predefined event types (process start and exit, function or block enter and exit, message send and receive, multicast communication, etc.), more general tracing is possible with user defined events. For this purpose, first an event identifier and the format string of the user event should be defined (similarly to the *printf* format strings in C). Then the predefined event format

can be used for trace event generation. Trace events can be generated by a function call with variable argument list that corresponds to the format definition.

3 Using GRM in the grid

The original structure of GRM can be used to collect trace data even if the application is running on a remote resource. In this case, the Local Monitors of GRM are running on remote hosts where the application processes are running.

The Netlogger toolkit [5] which has been used as – the only available – monitoring tool recently, has no local monitors. When monitoring with Netlogger, the processes should send data to the netlogd daemon to a remote site. This operation is very intrusive in a wide-area network environment. GRM Local Monitors provide a local shared-memory buffer for placing trace records as fast as possible bringing the intrusion to the minimum. Application processes thus, are not depending on the speed of the network.

GRM's structure works well for application monitoring but only if it can be set-up. In a local cluster or supercomputer environment the user can start GRM easily. The execution machine is known by the user and it is accessible, i.e., the user can start GRM on it manually. When GRM is set-up, the application can be started and it will be monitored by GRM. In the grid however, we do not know in advance which resource will execute our application. Even worse, even if we could know that (e.g. when selecting a given resource for its special purpose) we would not be able to start GRM manually on that resource. This is one of the main problems of all traditional tools designed for applications on local resources when moving towards the grid. One possible solution is to create a service from the local monitor and place it on all working nodes of all grid resources (or at least those that want to allow users to monitor their applications). This solution needs a local monitor that is able to run continuously and to handle the processes of different applications executed on the given working node one after the other.

First, we have chosen an alternative solution to minimize the reimplementation effort and keep the original GRM code to the maximum extent as possible. The code of the Local Monitor process has been turned into a library which is linked into the application — together with the instrumentation library. When the application process is started and calls the first instrumentation function, the instrumentation library forks a new process that becomes the local monitor. If there is already such a monitor (when more than one application process is launched on the same multi-processor machine) the application process connects to the already running instance. The local monitor connects to GRM's main monitor, creates the shared-memory buffer and initializes the connection between itself and the application process. After that, trace generation and delivery is done the same way as in the original local version.

Unfortunately, this solution works only if there are no firewalls between the local monitor and the main monitor processes that are running on different sites. As this is not the usual case, this version of GRM has a limited use. The firewall problem remains of course even for the first possible solution where the local monitor would be a permanent daemon on the working nodes. The only way to overcome this problem is

to introduce services that provide a proxy mechanism between the the user's machine and the working nodes of remote sites.

Another source of problems could be the use of forking for two reasons. First, some local job managers dislike jobs that fork out new processes, e.g. Condor which, in such a case, is not able to do check-pointing and clean-up when it is needed. Second, the forked out process duplicates the statically allocated memory areas of the original process. If the job has a large code and its static allocations are comparable to the available memory of the machine, it can cause memory allocation problems.

In the following section, our Mercury Monitor is introduced that solves both problems for application monitoring.

4 Mercury Monitor

The Mercury Grid Monitoring System [6] – being developed within the GridLab project [7] – provides a general and extensible grid monitoring infrastructure. Its architecture is based on the Grid Monitoring Architecture GMA [8] proposed by the Global Grid Forum (GGF). The input of the monitoring system consists of measurements generated by sensors. Sensors are controlled by producers that can transfer measurements to consumers when requested.

In Mercury all measurable quantities are represented as metrics. Metrics are defined by a unique name (such as, *host.cpu.user* which identifies the metric definition), a list of formal parameters and a data type. By providing actual values for the formal parameters a metric instance can be created representing an entity to be monitored. A measurement corresponding to a metric instance is called metric value. Metric values contain a timestamp and the measured data according to the data type of the metric definition. Sensors implement the measurement of one or more metrics.

The Mercury Monitor supports both event-like (i.e. an external event is needed to produce a metric value) and continuous metrics (i.e. a measurement is possible whenever a consumer requests it such as, the CPU temperature in a host). Continuous metrics can be made event-like by requesting automatic periodic measurements.

In addition to the components in the GMA, Mercury also supports actuators (similar to actuators in Autopilot [9] or manipulation services in OMIS [10]). Actuators are analogous to sensors but instead of taking measurements of metrics they implement controls that represent interactions with either the monitored entities or the monitoring system itself.

4.1 Architecture of Mercury Monitor

The Mercury Monitor components used for monitoring on a grid resource are shown in Fig. 2 drawn with solid lines. The figure depicts a grid resource consisting of three nodes. A Local Monitor (LM) service is running on each node and collects information from processes running on the node as well as the node itself. Sensors (S) are implemented as shared objects that are dynamically loaded into the LM code at run-time depending on configuration and incoming requests for different measurements. Requested information is sent to a Main Monitor (MM) service. The MM provides a central access

point for local users (i.e. site administrators and non-grid users). Grid users can access information via the Monitoring Service (MS) which is also a client of the MM. In large sized grid resources there may be more than one MM to balance network load. The modularity of the monitoring system also allows that on grid resources where an MM is not needed (e.g. on a supercomputer) it can be omitted and the MS can talk directly to LMs.

The elements drawn with dashed lines are involved in starting the application and not part of the monitoring infrastructure. The *resource broker* is the grid service that accepts jobs from grid users, selects a resource for execution and submits the job to the selected resource. The *jobmanager* is the public service that accepts grid jobs (e.g. GRAM in the Globus toolkit) for the grid resource. The LRMS is the local resource management system which handles job queues and executes jobs on the grid resource (e.g., Condor, PBS, LSF, etc.).

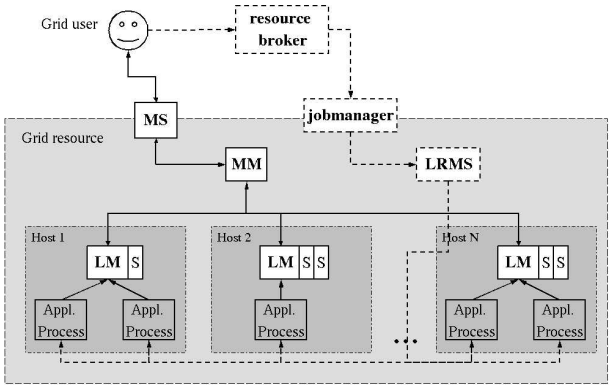

Fig. 2. Structure of the Mercury Monitor

As the Grid itself consists of several layers of services the monitoring system also follows this layout. The LM–MM–MS triplet demonstrates how a multi-level monitoring system is built by exploiting the compound producer—consumer entities described in the GMA proposal. Each level acts as a consumer for the lower level and as a producer for the higher level. This setup has several advantages. Transparent data conversion (such as, preprocessing or data reduction) can be easily supported by sensors at intermediate levels which get raw data from lower levels and provide processed data for higher levels. In cases when there is no direct network connection (e.g., because of a firewall) between the consumer and a target host, an intermediary producer can be installed on a host that has connectivity to both sides. This producer can act as a proxy between the consumer and the target host. With proper authentication and authorization policies at this proxy, this setup is more secure and more manageable than opening the firewall. These advantages are exploited in the MS which acts as a proxy between the grid resource and grid users. The MS is the component where grid security rules and local policies are enforced.

5 Integration of GRM and Mercury Monitor

When connecting GRM and Mercury Monitor (see Fig. 3) basically, the trace delivery mechanism of the original GRM is replaced with the mechanism provided by Mercury. The Local Monitors of GRM are not used now. The instrumentation library is rewritten to publish trace data using the Mercury Monitor API and to send trace events directly to the LM of Mercury Monitor.

The LMs of Mercury are running on each machine of the grid that allow monitoring of applications, so application processes can connect to an LM locally. Application monitoring data is considered to be just another type of monitoring data represented as a metric. To support application monitoring therefore, a sensor – providing this metric – called application sensor is created and the LM is configured to load it as a sensor module. The application sensor accepts incoming data from the processes on the machine, using the "extsensor" API of Mercury Monitor. Mercury uses metrics to distinguish different types of producers. GRM uses one predefined metric called `application.message` to publish data generated by instrumentation. This metric has a string data type and contains a trace event from the application process. Only the sensor providing the metric cares about the contents of the message, it is not processed within the components of Mercury Monitor.

The main process of GRM (called Main Monitor in the original version) now behaves as a consumer of Mercury, requesting the `application.message` metric from producers and subscribing to it. If there are such metrics (coming from application processes), Mercury Monitor delivers them to the main monitor of GRM via the channel created by requesting the metric and data streaming starts. As there may be several applications generating trace data at the same time, the `application.message` metric has an application ID parameter which is specified at subscription to identify the application from which trace information should be transferred for the given request.

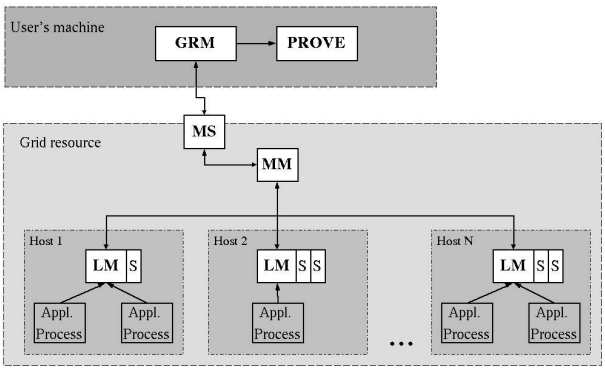

Fig. 3. Connection of GRM and Mercury Monitor on the grid

The use of the GRM tool as an application monitor has not been changed compared to the original usage. First, the application should be instrumented with GRM calls.

Then, the job should be started, which, in the case of a grid, means to submit the job to the resource broker (see boxes with dashed lines in Fig. 2), until the job is eventually started by the LRMS on a grid resource. Meanwhile, GRM can be started by the user on the local host that connects to Mercury Monitor and subscribes for trace information about the application. When the application is started and generates trace data, Mercury Monitor forwards the trace to GRM based on the subscription information.

5.1 Problems for application monitoring in the grid

As it was mentioned at the usage of the stand-alone GRM, application monitoring can be performed only if the monitoring architecture can be set-up at all. This architecture includes the applications themselves. Only when all components are at their place, connected together and know exactly what to do, application monitoring can be done. However, to be able to set-up this architecture for a given application that has just been submitted to the grid resource broker, the following problems should be solved:

- The address of the MS of Mercury Monitor on the grid resource, which executes the application, should be known by GRM to be able to subscribe there for tracing.
- GRM should identify the application in which it is interested in a way which is known both for GRM and Mercury.

Different grid resources have independent instances of Mercury Monitor. GRM should connect that grid resource which executes the actual job to be monitored. For the first problem, GRM has to find out, on which grid resource the job has started. Then, the public address of the Mercury Monitoring Service on that resource has to be found out. When these two pieces of information are available for GRM, it can connect to Mercury and subscribe for the trace.

For the second problem, the application should be identified to distinguish it from other applications and prevent mixing of different traces. Such distinction is already made for job handling in the grid by the resource broker, giving a global grid job id (GID) to each job. The user and thus, GRM can use this GID to identify the application. However, Mercury Monitor receives trace records from actual application processes that are identified with their process ID (PID) and a local user ID. During the steps of grid job submission from the user to the actual execution machines, the job gets different IDs such as, GID, GRAM job ID, LRMS ID and the processes get process IDs from the operating system. The relation between those IDs are not published by current grid middleware implementations. Thus, Mercury Monitor is not able to find out the mapping between processes and GID without cooperation from the middleware or the application. The authors of Mercury proposed a solution in [6] by creating a new temporary user account on the machines for each job. This solution has several advantages, however, it is not yet implemented in any grid projects.

The job ID mapping problem is an important problem to be solved in all grid implementations where monitoring, accounting and logging of applications is to be done in an accurate and safe way. Currently, the two different areas where GRM and Mercury is used for application monitoring, have two different solutions – rather to avoid the problem instead of solving it – with the same idea: if application processes can tell their corresponding GID to Mercury, the identification is done immediately.

In the EU-DataGrid (EDG) project [11], the middleware software components already propagate the GID of the job for logging purposes. The local resource manager starts a wrapper script before starting the actual job. This wrapper script does all necessary calls to middleware components on the grid resource before the job is executed. Moreover it also passes the global ID to the application process in an environment variable. GRM instrumentation library in the EDG version reads this environment variable and uses it as the application identifier. It should be mentioned, that this is not a safe solution as the process can rewrite the value of the environment variable before calling the first instrumentation call to GRM. Nevertheless, it is sufficient for monitoring cooperating applications but shouldn't be relied upon if monitoring data is used for accounting.

In the P-GRADE program development environment [1], the run-time system of P-GRADE assigns the global job ID to the application. This ID is used throughout the entire job execution to refer to the job (when attaching to the application within the user interface of P-GRADE, detaching from it, canceling the job, monitoring the job, etc.). As the code generation is in the hand of P-GRADE, the application processes can get this ID as a parameter at start and this parameter can be processed by the application-independent initialization code before the actual application code starts. This ID is reported by the GRM instrumentation library to Mercury Monitor as the identifier of the application and thus, Mercury does not need to care about the different IDs of the job as there is only one used.

The first task of finding the appropriate Mercury Monitor, however, cannot be solved without the help of external components. There may be as many solutions as many grid implementations but basically, the task is nothing else but the retrieval and the combination of two corresponding pieces of information: the execution resource and the address of the Monitoring Service on that resource. In the following section R-GMA, the information system of the EU-DataGrid project is shortly introduced and it is shown how it is used to solve this task of application monitoring on the EDG test bed.

6 R-GMA as an information system for GRM

R-GMA (Relational Grid Monitoring Architecture [12]) is the product of the EU-DataGrid project [11]. It is a GMA implementation based on the relational data model and using Java Servlet technology to provide both monitoring and information services. The special strength of the R-GMA implementation comes from the power of the relational model. It offers a global view of the information as if each Virtual Organization [13] had one large relational database. What R-GMA provides is a way of using the relational data model in a grid environment and it is not a general distributed RDBMS. All the producers of information are quite independent. It is relational in the sense that Producers announce what they have to publish via an SQL CREATE TABLE statement and publish with an SQL INSERT and that Consumers use an SQL SELECT to collect the information they need. There is a mediator in R-GMA that looks for the appropriate producers for a given query. It can also handle joint queries if data are archived in real databases and tables in the database are published to R-GMA (i.e. it lets execute the joint query by the database engine).

In theory, R-GMA as a monitoring system could also be used for transferring trace information to GRM instead of Mercury Monitor. However, our experiences showed that it is not well suited for application monitoring. The reasons for this lie within that R-GMA is built on web technology (servlets) and uses XML as a data encoding. A producer servlet is running in a central node of the grid resource and all trace records would need to be transferred from the application process (using a producer API of R-GMA) one by one, using HTTP protocol and XML encoding which has a high overhead. Such an infrastructure is appropriate to receive a relational record from each service and resource at a lower rate (say at every half a minute) and thus, provide resource monitoring but it is not able to provide the performance to transmit large trace data at a high rate with low overhead and intrusion, as required for on-line application monitoring. R-GMA as an information system however, can be used to solve the first task, i.e. to find the public address of the appropriate Mercury Monitoring Service.

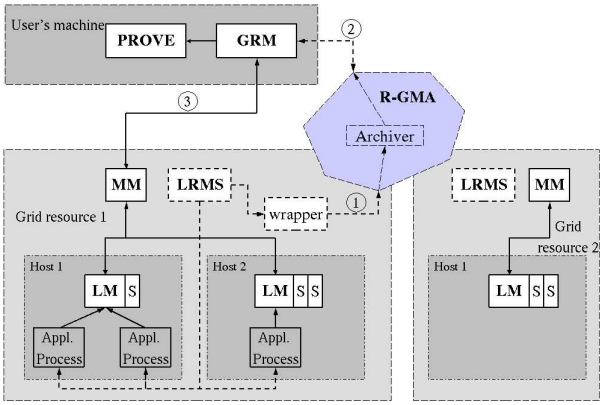

Fig. 4. Use of R-GMA by GRM and Mercury Monitor

The two required pieces of information could be published in two different relational tables, one used for publishing the GID of the job together with the name of the grid resource where it is executed. The second table could be used for publishing the URL of Mercury on a given grid resource together with the name of the resource. In this case, a joint query should be posed to R-GMA, looking for a given GID, joining the two tables on the name of grid resources and retrieving the URL of Mercury. This could be defined for R-GMA, however, a real database should be used for archiving those two tables to be able to join them. To avoid this dependency unnecessary for us, a simpler solution is used in the EDG test bed. The job wrapper of the EU-DataGrid software knows about the GID of the job and the front-end node of the actual grid resource for logging purposes. The MM of Mercury runs on this front-end node as well as all other public services of the middleware software. Thus, the wrapper script publishes the GID and the name of the front-end node together in one relational table (step 1 on Fig. 4). GRM poses a simple query to R-GMA to get records from this table for the given GID and retrieves the name of the front-end node (step 2). The MM uses a

predefined port address in the EDG test bed thus, combining the host name and the well-known port address, GRM can connect to Mercury Monitor right after getting the answer from R-GMA (step 3). Although the table is archived within R-GMA, it is just an in-memory archiver in a servlet somewhere and it is handled internally in R-GMA.

The connection of GRM, Mercury Monitor and R-GMA results in an application monitoring infrastructure that can be used for monitoring parallel applications running on a selected grid resource.

7 Conclusion and future work

The combination of GRM and Mercury Monitor provides a tool for monitoring message-passing applications that are executed in the grid. By using them, one can see and analyze on-line the behavior and performance of the application.

To test the performance of the monitoring system implementation we measured the time required to send GRM events from an instrumented application via the GRM sensor, the Mercury LM and MM to a consumer. These tests showed that the monitoring system is capable to handle more than 13000 GRM events per second. The time needed to transfer data was a linear function of the number of events. However, during development of Mercury so far the top priority was correctness and completeness and not performance, thus there are several areas for further improvements.

The current implementation of Mercury Monitor contains an application sensor that accepts trace data records from application processes through a UNIX domain socket. Writing trace records into a socket is about a magnitude slower action then writing into the memory directly because a context switch is required to the receiver process (the LM) which reads the record. In the original version of GRM, the Local Monitor of GRM created a shared-memory buffer and the application processes wrote trace records directly into that buffer. One of the most important tasks is to implement a new version of the application sensor to use shared-memory for tracing, achieving a better performance and bringing down the intrusion of monitoring.

The second problem is the scalability of applications in time. If an application is running for a long time, the generated trace information can be huge which is transmitted through the wide-area network. Usually, there is no need to monitor such long-running applications all the time. There is a need for the ability to turn monitoring off and on by the user (or even in the case of failures, slow-downs in network performance, etc.). The actuators in Mercury Monitor gives the possibility to do that from GRM. The application sensor will be stopped and restarted thus, stopping trace transmission outside of the machine.

The third problem is the scalability of applications in size. The architecture of the combination of GRM and Mercury is centralized which is not scalable to applications with a large size. This architecture supports the monitoring of a parallel application that runs on one grid resource but it cannot be used for large meta-computing applications running on several grid resources, except if it generates only as much trace information as an average parallel program. Even, a very large parallel application on a single but large cluster can generate so much trace data that cannot be delivered to another site to visualize. It is also questionable how the user looking for performance problems

can find useful information in huge amount of information. Researchers in the APART Working Group (Automatic Performance Analysis: Real Tools [14]) believe, that for such applications only automatic performance analysis tools can be used meaningfully. Also, trace data should be processed as close to the generation site as possible and to transfer only aggregate information upward to the performance tool and the user. The architecture of Mercury Monitor, the possibility to introduce new high-level sensors and actuators enables us to create a monitoring framework for automatic performance analysis.

References

1. P-GRADE Graphical Parallel Program Development Environment: http://www.lpds.sztaki.hu/projects/pgrade
2. Message Passing Interface Forum. MPI: A Message Passing Interface Standard, 1994
3. A.Geist, A.Beguelin, J.Dongarra, W. Jiang, B.Manchek and V.Sunderam. PVM: Parallel Virtual Machine – a User's Guide and Tutorial for Network Parallel Computing. MIT Press, Cambridge, MA, 1994.
4. Z. Balaton, P. Kacsuk, and N. Podhorszki. Application Monitoring in the Grid with GRM and PROVE., Proc. of the Int. Conf. on Computational Science, ICCS 2001, San Francisco, pp. 253-262, 2001
5. B. Tierney et al. The NetLogger Methodology for High Performance Distributed Systems Performance Analyser. Proc. of the IEEE HPDC-7 (July 28-31, 1998, Chicago, IL), LBNL-42611
6. Z.Balaton, G.Gombás. Resource and Job Monitoring in the Grid. Proc. of EuroPar'2003 Conference, Klagenfurt, Austria, pp. 404-411, 2003
7. GridLab project home page: http://www.gridlab.org
8. B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. A grid monitoring architecture. GGF Informational Document, GFD-I.7, GGF, 2001, URL: http://www.gridforum.org/Documents/GFD/GFD-I.7.pdf
9. R. Ribler, J. Vetter, H. Simitci, D. Reed. Autopilot: Adaptive Control of Distributed Applications. Proc. 7th IEEE Symposium on High Performance Distributed Computing, Chicago, Illinois, July 1998.
10. T. Ludwig, R. Wismüller. OMIS 2.0 - A Universal Interface for Monitoring Systems. Proc. 4th European PVM/MPI Users' Group Meeting, Cracow, Poland, November 1997.
11. EU-DataGrid project home page: http://www.eu-datagrid.org
12. S.Fisher et al. R-GMA: A Relational Grid Information and Monitoring System. 2nd Krakow Grid Workshop, Krakow, Poland, 2002
13. I. Foster, C. Kesselman, S. Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications, 15(3), 2001.
14. APART Working Group: http://www.fz-juelich.de/apart-2/